VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Nástroj pro správu metadat ORM rámce

Tool for a Maintanance of ORM Data

2018 Jiří Klemš

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jiří Klemš**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Nástroj pro správu metadat ORM rámce
Tool for a Maintanance of ORM Data

Jazyk vypracování: čeština

Zásady pro vypracování:

Firma ITeuro, a.s. pro vývoj aplikace využívá vlastní implementaci objektově relačního mapování (ORM), které pracuje s metadaty uloženými v databázi. Hlavním cílem této práce je vytvořit grafické rozhraní pro práci s metadaty. Mezi hlavní funkce výsledné aplikace bude patřit:
1. Možnost definovat a upravovat nové datové zdroje v metadatech.
2. Aplikace bude podporovat datové zdroje typu tabulka, pohled a procedura/funkce.
3. U datových zdrojů bude možné definovat vlastnosti odpovídající všem datovým typům a integritním omezením standardně nabízených v SQL Serveru.
4. Exportovat a importovat metadata z/do XML souboru.

Práce bude probíhat v následujících krocích:
1. Rešerše ORM rámců, které se běžně používají v prostředí .NET.
2. Srovnání standartních ORM rámců s řešením použitým ve firmě ITeuro, a.s.
3. Analýza, návrh, implementace a testování nástroje splňující výše uvedené požadavky.

Seznam doporučené odborné literatury:

[1] Dokumentace k etity framework. URL: https://docs.microsoft.com/en-us/ef/
[2] Dokumentace k NHibernate, URL: http://nhibernate.info/doc/index.html

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Radim Bača, Ph.D.**

Datum zadání: 01.09.2017

Datum odevzdání: 30.04.2018

doc. Ing. Jan Platoš, Ph.D.
vedoucí katedry

prof. Ing. Pavel Brandštetter, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2018 .....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 30. dubna 2018 .....

Na tomto místě bych rád poděkoval Ing. Radimu Bačovi, Ph.D. za poskytnutí užitečných rad při tvorbě této práce. Poděkování si zaslouží i moji přátelé a rodina za obrovskou dávku trpělivosti.

Abstrakt

Tato bakalářská práce se za zabývá porovnáním běžně dostupných objektově relačních mapovacích rámců na platformě .NET s možností napojení na SQL server. Úvodní část představuje existující ORM řešení, jejich výhody a nevýhody použití. Další část se věnuje popsání klíčových způsobu mapování a srovnání s vlastním řešením používaným ve firmě ITeuro a.s. Hlavní zaměření je pak kladeno na vytvoření aplikace pro snadnější a rychlejší tvorbu mapovacího objektu pro střední vrstvu využívanou aplikací.

Klíčová slova: ACID, bakalářská práce, NHibernate, LINQ to SQL, ORM, Entity Framework, XML, .NET, SQL, datový zdroj

Abstract

This bachelor thesis deals with comparison of commonly available object relational mapping frames on the .NET platform with the possibility to connectit to the SQL Server. The first part introduces existing ORM solutions, their advantages and disadvantages. The next part deals with the description of the key ways of mapping and comparing this solution to ITeuro a.s. solution. The main focus is on creating application for easier and faster object mapping for the middle layer used by the application.

Key Words: ACID, bachelor thesis, NHibernate, LINQ to SQL, ORM, Entity Framework, XML, .NET, SQL, data source

Obsah

Seznam použitých zkratek a symbolů **8**

Seznam obrázků **9**

Seznam tabulek **10**

1 Úvod **11**

2 Objektově relační mapování **12**
2.1 Výhody použití ORM 12
2.2 Nevýhody použití ORM 12
2.3 Vlastní ORM rámec 13

3 Porovnání vybraných ORM rámců **15**
3.1 NHibernate 15
3.2 Entity Framework 18
3.3 LINQ to SQL 22
3.4 Nevýhody současných řešení 24

4 Vlastní řešení **26**
4.1 Obecný popis řešení 26
4.2 Definice pojmů datového zdroje 27
4.3 Funkční analýza 33
4.4 Konceptuální model 36
4.5 Použití datového zdroje v aplikaci 37

5 Závěr **42**

Literatura **44**

Seznam použitých zkratek a symbolů

ACID	–	vlastnost databázových transakcí: A - atomičnost, C - korektnost, I - izolovanost a D - trvalost
API	–	Application Programming Interface
CSDL	–	Conceptual Schema Definition Layer
C#	–	Object oriented language
DDL	–	Data Definition Language
DML	–	Data manipulation language
EDM	–	Entity Data Model
EF	–	Entity Framework
EFD	–	Entity Framework Designer
HQL	–	The Hibernate Query Language
LINQ	–	Language Integrated Query
POCO	–	Plain Old CLR Object
ORM	–	Object-Relational Mapping - objektově-relační mapování
RPC	–	Remote procedure call
SSDL	–	Storage Schema Definition Layer
T-SQL	–	Transact-SQL
VB.NET	–	Visual Basic for .NET
WinFS	–	Windows Future Storage
XML	–	eXtensible Markup Language – rozšiřitelný značkovací jazyk

Seznam obrázků

1 Architektura NHibernate [5] 15
2 Stav objektů v NHibernate [8] 17
3 Architektura Entity Framework [7] 19
4 Architektura LINQ to SQL [17] 23
5 Konceptuální model 41

Seznam tabulek

1 Spárování vlastnosti s navráceným resultem z vlastní metody načtení dat 31
2 Poskládání dotazu z definice datového zdroje 39

1 Úvod

Většina komplexnějších aplikací potřebuje nějaký způsobem řešit perzistentní uložení dat. V posledních 40 letech od svého vzniku získaly pro uchování dat na oblibě relační databáze postavené nad relačním modelem. Jejich využití a rozšíření neovlivnil ani pozdější příchod jejich plánovaného nástupce v podobě objektových databází nebo následného kompromisního řešení v podobě objektově relačních databází. Pro menší aplikace nebo různé typy projektů je možné použit i například XML nebo NoSQL databáze, které sice nabízejí pro některé operace rychlejší práci s daty, ale je to mnohdy na úkor zachování konzistence dat nebo podpory ACID modelu. Proto je také vhodné si před tvorbou aplikace promyslet, k jakému účelu bude sloužit a zvolit tak ekvivalentní úložiště.

Evoluce vývoje programovacích jazyků byla razantnější než u databázových systému. Přechod od procedurálního k objektovému programování se stal v současnosti rozšířeným standardem pro vytváření nejen rozsáhlých desktopových aplikací, ale i aplikací webových. Při tvorbě bakalářské práce vznikla potřeba řešit komplikovaný převod objektů do relací. Pro převod může programátor použít vlastní vytvořené řešení, nebo využít některých z již dostupných rámců pro příslušnou platformu. Převod však nebývá vždy bezproblémový a je označený souhrnným názvem impedance „mismatch“. Je nutné vytvořit mapovací objekty a funkce, které umožní přímé mapování objektů na databázové objekty typu tabulka a pohled a vývojář pak může pracovat přímo s objekty bez hlubší znalosti problematiky relačních databází. Práce je poté mnohem jednodušší a komfortnější. Pro vlastní tvorbu mapování se dají použít některé návrhové vzory (Table Data Gataway, Row Data Gataway, Active Record nebo Data Mapper), které definoval Martin Fowler [1].

Teoretická část bakalářské práce je zaměřena na představení ORM. Popisuje danou techniku, zmiňuje problémy, její výhody při použití, ale zároveň také nevýhody. Představeny jsou nejčastěji používané mapovací rámce pro platformu .NET včetně popisu, jak každý z nich přistupuje k dané problematice a techniky využívající pro převod.

Navazující praktická část bakalářské práce je zaměřena na mapování, které se ve společnosti ITeuro, a. s. používá a popisuje odlišnosti a potřeby pro vznik vlastního řešení. Na příkladech jsou uvedena odlišná řešení použití při vlastní implementaci oproti běžně dostupným nástrojům na trhu. V této části je důraz kladen především na analýzu, návrh a implementaci grafického nástroje, který bude sloužit k zrychlení a efektivnějšímu vývoji datových zdrojů pro aplikaci. Mezi stěžejní funkce bude patřit možnost definovat a upravovat nejen nové datové zdroje, ale i možnost exportu a importu do XML souboru, který se používá pro přenos mezi různými systémy.

V závěrečné části budou popsány problémy, se kterými jsem se během tvorby a při testovaní aplikace setkal, dále pak postřehy pro následující vývoj a rozšíření aplikace.

2 Objektově relační mapování

Při tvorbě aplikací často řešíme potřebu ukládání dat. K tomu se nejčastěji využívají relační databáze a pro vývoj aplikací objektově orientované jazyky. V tomto případě ale dochází ke střetu dvou rozdílných konceptů. Objektové modelování popisuje systém pomocí objektů, které představují určitou abstrakci a mají svoji identitu, chování a stav. Naproti tomu relační model popisuje sytém pomocí informací. [2]

2.1 Výhody použití ORM

Použití ORM mapování přináší programátorovi několik výhod:

- Nezávislost na datovém uložišti - s menšími úpravami je možné snadno vyměnit databázové uložiště dat za předpokladu dodržení struktury mapování. To sebou nese i výhodu při testování nebo nasazení aplikace do produkčního systému nebo přenosu na jinou platformu. Mimo to programátor nemusí znát detailně specifikaci odpovídající dané databázi. To za něj řeší příslušný použitý mapovací rámec.

- Způsob přístupu k datům - práce s objekty jakožto reprezentanty reálného světa je bližší vývojáři, než práce s n-ticemi. Pro přístup k datům se v rámci ORM používají objekty, takže vývojář je izolovaný od skutečného uložení dat, které nemusí být uloženy nakonec ani v databázi, ale například v souboru. Nedílnou součástí je také oddělení datové vrstvy od aplikační vrstvy.

- Náklady na vývoj - při seznámení se s příslušným rámcem pro mapovaní a jeho zvládnutí, lze dosáhnout zvýšení produktivity při tvorbě aplikace. Vývojář nemusí psát obslužný kód pro ukládání nebo modifikaci dat v databázi. Toto za něj obstará mapovací funkce, která dokáže poskytnout i opačnou podporu, tedy pokud dojde ke změně obsahu dat v databázi, tak se změny projeví i v příslušném objektu, který s nimi pracuje.

  ORM kód generovaný rámcem bývá zpravidla kratší, vede k eliminaci zbytečně se opakujících části kódu. Naproti tomu přístup k datům zapsaný pomocí SQL dotazu je sice vykonáván rychleji, ale je potřeba ze strany vývojáře napsat ošetřující kód pro vykonání dotazu a jeho zpracování. Chybná syntaxe kódu jazyka SQL, který je uložený v podobě řetězce, je vrácena až za běhu aplikace.

- Čitelnost zdrojového kódu - kód napsaný pomocí ORM mapování je kratší, čitelnější a zaměřený na aplikační logiku programu místo toho, aby řešil přístupy a zpracování dat pomocí SQL. Další výhodou je snadnější udržitelnost zdrojového kódu.

2.2 Nevýhody použití ORM

- Vytvoření a udržování mapování - pro správné fungování ORM je potřeba vytvořit odpovídají vrstvu, která je ekvivalentní se strukturou tabulek, pohledů v databázi. V případě

rozsáhlých databází obsahujících velké množství těchto entit, může být ruční vytvoření mapování náročné a i náchylné k chybám. Proto některé pokročilejší rámce obsahují funkcionalitu pro automatické vytvoření mapování na základě námi vybraných databázových objektů. Problém může nastat, pokud dojde ke změně databázové struktury (přidání nových sloupců, tabulek, změna datového typu u sloupce). V tomto případě je nutné upravit i mapování, nebo pokud rámec funkcionalitu podporuje, tak provést aktualizaci mapovacího modelu. Bohužel ne vždy tato automatická aktualizace proběhne korektně a je potřeba pak provést ruční úpravu.

- Snížení výkonu - použitím ORM přidáváme do aplikace další vrstvu, která vede nevyhnutelně ke zpomalení aplikace. To se dá částečně eliminovat například přidáním mezipaměti. Dalším problém nastává při dotazu na strukturu 1:N, kde se pro daný objekt načítají další potřebné objekty. Ten se dá obejít pomocí návrhového vzoru *Lazy Load*, , kde se pro objekt načítají další objekty až v případě, pokud je na ně vznesen požadavek. ORM také není vhodné pro *bulk operace*, například pro větší hromadné vkládání dat.

- Rozdílný způsob dotazů do databáze - různé ORM rámce používají svůj specifický jazyk pro dotazy do databáze. Je potřeba se seznámit s tímto jazykem a je také potřeba znát i alespoň základní syntaxi jazyka SQL. Většina jich ale podporuje dotazování pomocí LINQ, takže přechod nebo použití různých mapovacích rámců je pak snadnější, protože již není potřeba se seznamovat s dalšími jazyky.

- Počáteční náročnost a náklady - vývojář se musí na začátku seznámit s patřičným mapovacím rámcem, jeho specifikací, aby byl schopný vytvořit příslušný ORM model a správně ho nastavit. V případě potřeby ho i upravit podle vzniklých změn či požadavků.

- Kontrola nad SQL dotazy - ORM rámec by měl sice na pozadí generovat optimalizované SQL dotazy, ale nemusí tomu vždy být. Naproti tomu při ruční tvorbě dotazů má programátor větší míru kontroly a může případný dotaz optimalizovat například vynucením použití konkrétního indexu, nebo úpravy spojení. Pro kontrolu zaslaného dotazu z aplikace existují různé trasovací nástroje, kde je vidět zaslaný dotaz a můžeme ho vzít a podívat se na jeho exekuční plán.

2.3 Vlastní ORM rámec

Stává se, že je vývojář postaven před rozhodnutí, zda některý z existujících rámců je použitelný právě pro aplikaci, která se má vytvořit. V současné době většina rámců poskytuje velké množství funkcionalit, které se používají, ale také nemusí podporovat nebo nabízet prostředky, které bychom zrovna potřebovali. V takovém případě je vhodné se zamyslet nad vytvořením vlastního mapování.

Při vytváření vlastního rámce máme možnost přizpůsobit každý specifický dotaz našim požadavkům a tak i zjednodušit zpracování komplexnějších problémů. Příkladem takového ná-

ročnějšího řešení by mohlo být, pokud bychom se rozhodli použít jako zdroj dat pro vlastnost na formuláři databázovou funkci řešící komplikovanější výpočet. Při návrhu můžeme zvolit dva způsoby:

- Vytvoření obecného kódu, který dokáže pracovat s libovolným objektem bez předchozí znalosti jeho obsahu. Tato varianta nese sebou využití i u dalších případných projektů, ale v některých případech může mít horší výsledky při zpracování některých požadavků.
- Pro každý specifický požadavek vytvořit vlastní mapovací funkce, které lze ale využít jen pro konkrétní požadavek. To může přinést rychlejší zpracování dotazů, ale může být zdrojem nepřehlednosti a chybovosti v kódu.

Nevýhodou při tvorbě vlastního řešení může být potřeba znalostí ORM technik, případných návrhových vzorů (Table Data Gataway, Row Data Gataway, Active Record nebo Data Mapper) [1], které se při vývoji dají použít. Popsat problémy, na které můžeme narazit při vytváření vlastního řešení, je značně komplikované, protože se projeví až při jeho tvorbě. Vlastní přístup však ale může způsobit mnoho potíží, jako jsou zbytečné složitosti v kódu, nepřehlednost a neoptimalizované dotazy.

Pro vytvoření vlastního rámce vedla myšlenka k rychlé a snadné možnosti provádění úprav, rozšíření a přenositelnosti mezi jednotlivými testovacími a produkčními systémy a umožnit tak zákazníkům snadné vlastní rozšíření stávajícího programu. Dalším účelem vzniku aplikace je i přidáni možnosti snadnější kontroly a rychlého dohledání problému vývojáři. Výhodou takto vytvořeného datového zdroje je, že není součástí programu, ale je uložený ve vlastní databázi, odkud se při použití datového zdroje v programu načte příslušná definice a sestaví se z ní za běhu datový zdroj, který je pak použitý. Podrobnější popis bude následovat v kapitole 4.

3 Porovnání vybraných ORM rámců

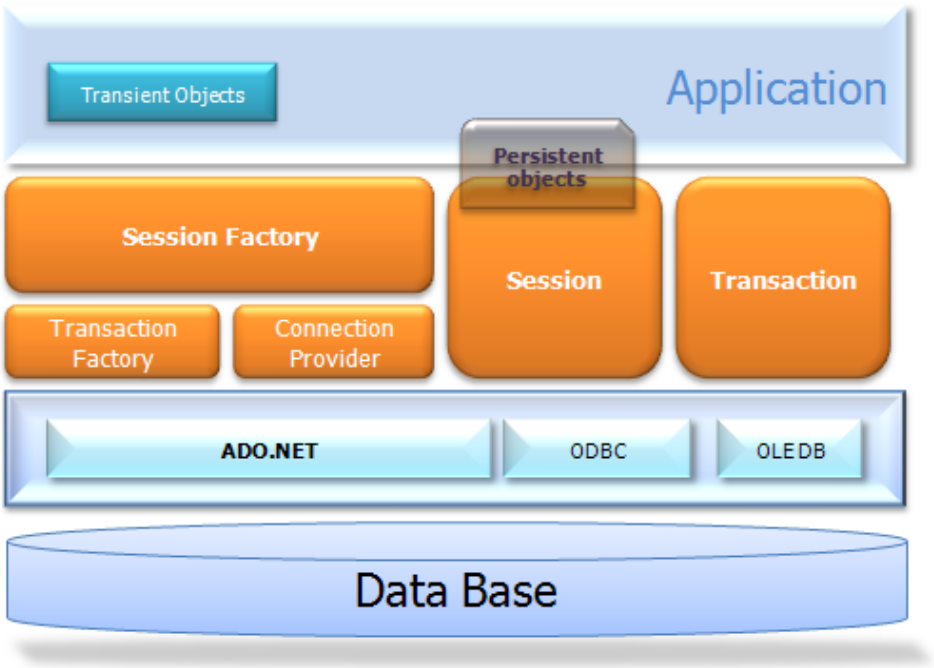
3.1 NHibernate

NHibernate je open-source ORM rámec pro platformu .NET, který vznikl jako port z již existujícího ORM Hibernate verze 2.1 [3] napsaného v jazyce Java.

3.1.1 Historie

S vývojem začal Tom Barrett a později se k němu připojil Mike Doerfler a Peter Smulovics. Na konci roku 2005 se společnost JBoss, Inc., stala součástí firmy Red Hat a byl přijmut Sergey Koshcheye jako hlavní vývojář pro NHibernate na celý úvazek pro dokončení portu a práci na dalších verzích. Na konci roku 2006 byl ukončený vývoj a podpora tohoto projektu. Ten následně převzala komunita pro další vývoj [4]. Od verze 5.0 byla přidána podpora pro asynchronní programování. Aktuální verze 5.0.1 byla uvolněna 13. 11. 2017. Stažení lze provést přímo ze stránek komunity nebo doinstalovat do projektů pomocí Package Manager Console pomocí příkazu: *Install-Package NHibernate -Version 5.0.1.*

3.1.2 Architektura



Obrázek 1: Architektura NHibernate [5]

NHibernate poskytuje několik možností konfigurace a přístupu do datového uložiště. Mezi základní komponenty patří:

- Rozhraní **ISessionFactory** - je zkompilovaný neměnný mapovací objekt pro jednu databázi. Používá návrhový vzor Factory a slouží pro vytváření objektů typu `ISession`. Může být také využit volitelně pro mezipaměť druhé úrovně.
- Rozhraní **ISession** - je jednoduchý objekt sloužící pro komunikaci mezi aplikací a trvalým uložištěm. Obaluje technologii ADO .NET pro připojení. Uchovává povinně první úroveň mezipaměti trvalých objektů. Takto vytvořený objekt není možné sdílet mezi více vláknovými aplikacemi.
- Rozhraní **ITransactionFactory** - jedná se o volitelný objekt. Slouží k vytvoření instancí transakcí. Může být vývojářem implementován nebo rozšířen.
- Rozhraní **ITransaction** - jde o objekt používaný aplikací pro specifikaci atomických operací. Odstiňuje aplikace od používání transakcí v ADO .NET. V rámci instance `ISession` objektu může být obslouženo více transakcí.
- **IConnectionProvider** - používá návrhový vzor Factory pro vytvoření připojení a příkazů ADO .NET. Představuje abstrakci od dodaných implementací `DbConnection` a `DbCommand`.

3.1.3 Mapování

NHibernate ve výchozí instalaci neobsahuje žádný grafický nástroj pro automatické vytvoření mapovacího modelu z již existujícího databázového uložiště. Existují ale programy třetích stran, které umožňují automatické vytvoření mapovacího modelu a tak pokud databáze obsahuje velké množství tabulek a pohledů, její vytvoření nemusí být až tak náročné.

Pro menší množství databázových objektů je ruční vytvoření mapování poměrně rychlá záležitost. Pro každou tabulku musí být vytvořena vlastní třída, kterou bude popsaná. NHibernate podporuje POCO (Plain Old CLR Object) objekty, pro které ale vyžaduje, aby měly bezparametrický konstruktor a deklarované vlastnosti pro sloupce mapované tabulky.

Mapování mezi tabulkou a třídou je popsáno v XML souboru s ekvivalentním názvem třídy a tabulky. Například pokud máme třídu a tabulku s názvem „Polozka“, tak mapovací soubor bude pojmenován Polozka.**hbm.xml**

Kardinalita vztahu

Kardinalitu mezi objekty mapovanými na tabulku je možné vytvořit pro všechny tři základní typy 1:1, 1:M a M:N a to oběma směry. Násobnost se zapisuje v mapovacím XML pomocí elementu <one-to-one>, <one-to-many> nebo <many-to-one> a <many-to-many>.

Pro mapování dědičnosti je možné využít 3 návrhové vzory:

- table per class hierarchy
- table per subclass

- table per concrete class

Navíc NHibernate přidává ještě 4 mírně odlišný typ polymorfismu:

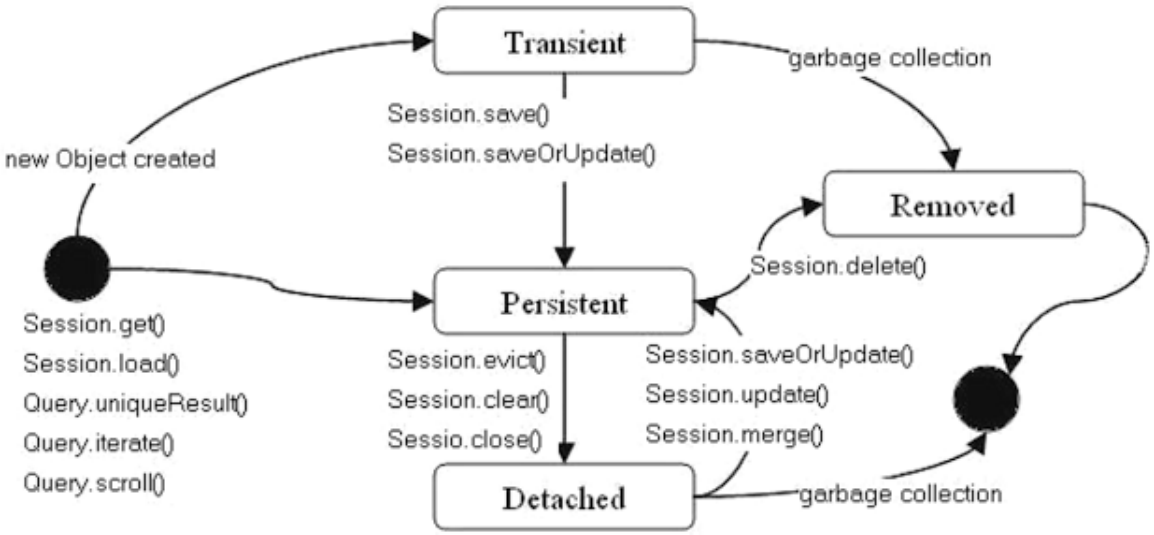
- implicit polymorphism

Objekty

Vytvořené objekty mohou být ve 4 stavech. Jedna instance objektu odpovídá jednomu řádku tabulky z databáze. Po vytvoření je objekt v dočasném stavu (Transient). Není spojený s žádnou Session a nachází se jen v paměti. Změny provedené v tomto objektu nejsou sledované.

Pokud objekt vytvoříme v asociaci se Session a uchovává data z tabulky, hovoříme o trvalém objektu (Persistent). Je možné ho použít i po zavření Session.

Další skupinou jsou objekty odpojené od Session, které už jsou uložené v databázi. Ty nazýváme (Detached). Hlavním rozdílem oproti objektům v trvalém stavu je, že se nedá zajistit, jestli data objektu v paměti odpovídají záznamu v databázi.



Obrázek 2: Stav objektů v NHibernate [8]

3.1.4 Možnosti dotazování

Pro dotazování a manipulaci s daty v databázi nabízí NHibernate několik možností:

- **HQL** - jedná se o objektový dotazovací jazyk který je case-insensitive, podobný jazyku SQL. Podporuje spojení, agregace, pod dotazy, seskupování. Umožňuje vytvořit parametrické dotazy, kde se parametr označuje dvojtečkou a je možné ho později přiřadit. Přístup lze získat pomocí instance `ISession.CreateQuery()`.

- **Criteria Queries** - část Criteria API. Jde o objektový dotazovací jazyk, který umožňuje nad takto vytvořeným objektem aplikovat filtrační pravidla a logické podmínky. Tento způsob zápisu je oproti předchozím hůře čitelný a nenabízí kontrolu vlastností při překladu, které jsou zapsané textově. Od verze NHibernate 3.0 (.NET 3.5) byla přidána podpora QueryOver API. To přináší typově statickou kontrolu a obal okolo původního API. Zavádí podporu rozšiřitelných metod a lambda výrazů. Umožňuje vracet jen požadované sloupce místo celé entity.
- **LINQ to NHibernate** - byl představený ve verzi 3.0 a umožňuje používat LINQ API pro dotazování.
- **Native SQL** - umožňuje vytvářet dotazy v syntaxi jazyka SQL a používat různá databázová rozšíření jako například v případě MSSQL serveru querry hint. Dále přináší možnost napsaní vlastních uložených metod pro DML příkazy. Je potřeba vytvořit instanci `ISession.CreateSQLQuery()`.

3.2 Entity Framework

Entity framework (dále jen EF) je open - source ORM rámec pro ADO .NET od firmy Microsoft. Dříve byl součástí .NET frameworku, ale od verze 6.0 je od něj oddělený.

3.2.1 Historie

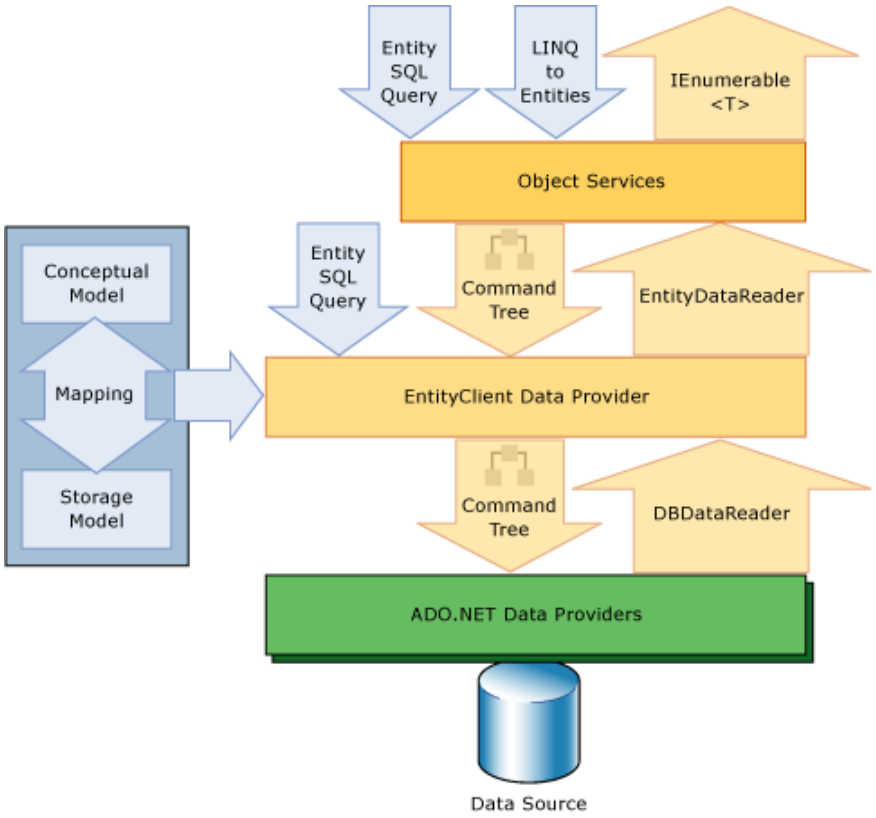
První verze Entity Frameworku (EF) byla vydaná 11. srpna 2008 jako součást .NET Frameworku SP1 a Visual Studio 2008. Ta ale byla přijata od vývojářů se značnou kritikou. Další verze označené EF 4 byla vydaná současně s .NET 4.0 v dubnu 2010 a řešila značné množství připomínek z předešlé verze. Třetí verze označená EF 4.1 přinesla podporu mapování Code First a byla zveřejněna 12. dubna 2011. EF 5 byl uvolněný společně s .NET frameworkem 4.5. Následující verze 6.0 byla vydána 17. října 2013 už jako open-source projekt. Po té vývoj pokračoval verzí EF 7, ale ta byla přejmenována na Entity Framework Core 1 (EF Core 1.0) a byla uvolněna společně s ASP.NET Core 1.0 27. června 2016. Aktuální verze EF Core 2.0 byla představena společně s Visual Studio 15.3 a ASP.NET Core 2.0 14. srpna 2017. [6]

3.2.2 Architektura

Entity Data Model (dále jen EDM) je rozšířenou verzi Entity-Relationship modelu, který specifikuje konceptuální model dat pomocí různých modelovacích technik. EMD podporuje množinu základních datových typů určených pro popis vlastností a je uložený v XML souboru s příponou edmx. Skládá se ze tří modulů:

- **Storage Schema Model** - někdy také označovaná jako Storage Schema Definition Layer (SSDL). Obsahuje model databáze, tabulky, pohledy, uložené procedury, vazby mezi nimi a klíče. Může být vytvořena na základě již existující databáze nebo z konceptuálního modelu.

- **Conceptual Model** - někdy označený jako Conceptual Schema Definition Layer (CSDL) odpovídá entitnímu modelu. Na tuto vrstvu se posílají dotazy.
- **Mapping Model** - mapovací vrstva zpracovává pouze propojení mezi konceptuálním a datovým modelem.



Obrázek 3: Architektura Entity Framework [7]

Visual studio obsahuje nástroj nazvaný Entity Designer určený pro tvorbu EDM.

3.2.3 Mapování a tvorba modelu

EF nabízí tři základní možnosti jak vytvořit EDM. Každý z nich má své výhody a nevýhody:

- **Code First** - tento model tvorby EDM použijeme v případě, že databáze neexistuje, anebo neobsahuje tabulky. Nejdříve se musí vytvořit POCO třídy, podle kterých se vygeneruje datový model. Třídy je možné definovat v jazyku C# nebo VB.NET.
- **Model First** - tuto variantu je vhodné použít při vytváření nového projektu v době, kdy návrh databázové struktury neexistuje. Model je uložený v edmx souboru a může být spravován pomocí entity framework designeru (EFD). Na základě modelu pak EFD vygeneruje SQL skripty pro vytvoření databáze.

- **Database First** - představuje alternativu k předchozím dvěma variantám. Zde se EDM vytváří z již existující databáze. Při vytváření je dána možnost si vybrat, které databázové objekty chceme, aby byly zahrnuty do modelu.

Entita

Entita je vytvořená instance EMD a je nazývaná jako Entity Type. Představuje základní prvek konceptuální vrstvy. Entita má vlastnosti podobné vlastnostem objektu. Existuje více typů:

- **EntityObject** - jde o základní entitní třídu generovanou entity data modelem a proto jsou závislé na EF.
- **POCO (Plain Old CLR Objects)** - jsou třídy nezávislé na rámci. Reprezentanty jsou prosté třídy z .NET.
- **POCO proxy** - jedná se o obalené POCO třídy, které musí splňovat určité podmínky. Musí být veřejné, nesmí být abstraktní, sealed, konstruktor musí být bezparametrický. Nesmí implementovat rozhraní `IEntityWithRelationships` a `IEntityWithChangeTracker`. Umožňuje využití lazy load a change tracking proxies.
- **Self-Tracking Entities** - využijí se ve vícevrstvých aplikacích, kde jsou většinou odpojeny od kontextu a musíme nějakým způsobem sledovat provedené změny a předat je zpět do kontextu a uložit je. Pro sledování změn se musí implementovat rozhraní `IObjectWithChangeTracker` a `INotifyPropertyChanged`. Neumožňují použití lazy load.

Kardinalita vztahu

Kardinalita je další ze základních bloků sloužící k popisu vztahů v EDM. V konceptuálním modelu představuje vztah mezi dvěma entitami a je charakterizována násobností. Násobnosti mohou být typu 1:1 je i podporována násobnost 0..1, 1:M a M:N. Kardinalita může být typu:

- vztah cizího klíče - mezi dvěma tabulkami a entitami musí být definována vazba cizího klíče.
- nezávislý vztah - chybí vazba cizího klíče.

Object Services

Jedná se o nejhornější vrstvu EF. Stará se o transformaci dat získaných z databáze do objektů. Umožňuje sledování změn v objektech, jejich uložení a použití lazy load. Podporuje jak dotazování pomocí LINQ tak i Entity Querry.

Práce s entitami může být dvojího typu:

- **Connected Scenario** - jde o stav, kdy je entita načtena z databáze a v rámci stejného kontextu je i uložena zpět.
- **Disconnected Scenario** - entita je načtena v rámci jednoho kontextu, ale může dojít k modifikaci a jejímu vrácení a uložení zpět do databáze v jiném kontextu.

3.2.4 Možnosti dotazování

Pro dotazování jsou k dispozici dvě možnosti:

- **Entity SQL** - jde o jazyk syntaxí podobný SQL, který je nezávislý na databázovém úložišti a pracuje přímo s entitami. Hlavní odlišností od jazyka SQL je podpora dědičnosti a vztahu z EDM, nepodporuje však transakční zpracování např. DML a DDL operace, rozšíření klauzule group by a query hint. [9]

  Vhodné použití je v následujících případech:

  - pokud chceme vytvořit dynamický dotaz sestavený za běhu, použijeme `ObjectQuery<T>`.
  - definujeme dotaz jako součást datového modelu. K tomu je potřeba použít QueryView Element (MSL). [10]
  - při použití EntityClient pro navrácení řádků z entity pouze pro čtení pomocí `EntityDataReader`. [11]
  - při znalostech SQL jazyka může být přirozenou volbou. [12]

- **LINQ to Entities** - jde o rozšíření jazyka LINQ, kde dotaz zapsaný v jazyku LINQ je pomocí LINQ to Entities převeden do výrazového stromu a ten se pak následně provede vůči kontextu. Vrácený objekt může být použit buď EF nebo LINQ.

  Postup pro vytvoření a spuštění LINQ to Entities dotazu: [13]

  - vytvoření instance `ObjectQuery <T>` z `ObjectContext`.
  - vytvoření dotazu LINQ to Entities buď v jazyce C# nebo Visual Basic pomocí instance `ObjectQuery<T>`.
  - převede standardní LINQ operátory a výrazy do příkazového stromu.
  - vykonání dotazu v podobě výrazového stromu proti zdroji dat. Pokud dojde k vyvolání výjimky během provádění dotazu, tak je vrácena ihned zpět na klienta.
  - vrácení výsledku dotazu zpět na klienta

Pro dosažení lepšího výkonu a odstranění opakované režie s kompilací LINQ dotazu byla od verze EF 5.0 zavedena třída `CompiledQuery`. Ta umožnuje provést překlad jen při prvním spuštění dotazu a pak už vrací jen delegáta, který se odkazuje přímo na zkompilovaný dotaz do cache paměti EF. Tato vlastnost se nazývá `Auto-Compiled LINQ Queries` a ve výchozím stavu je zapnutá, ale dá se vypnout. [14]

LINQ to Entities podporuje seskupování, agregace, stránkování a řazení. Pokud potřebujeme provést přidání, změnu nebo smazání v rámci entity, změna se provede zavoláním metody `SaveChange()` na kontextu.

3.3 LINQ to SQL

LINQ to SQL je pravděpodobně prvním ORM nástrojem od firmy Microsoft. Jde tedy o jedno z mnoha rozšíření jazyka LINQ, které lze využít pro získání dat jen z MS SQL databáze.

3.3.1 Historie

Jazyk LINQ byl představený 19. listopadu 2007 společně s verzí .NET frameworku 3.5. Samotný začátek vývoje sahá až do roku 2003. Původní potřebou bylo vytvoření nástroje pro práci s výrazovými stromy a překladače jazyka SQL. Měl být zakomponovaný do ObjectSpace a části C-Omega. [15]

Velká část vývoje byla v té době směřována k vývoji WinFS. Později byl vývoj WinFS zrušený a ObjectSpace nebyl ve stavu, aby se mohl stát součástí .Net Frameworku 2.0. Nakonec bylo rozhodnuto, že se LINQ to SQL stane ORM nástrojem pro SQL server. Jeho vývoj byl ukončen v roce 2008. [16]

3.3.2 Architektura

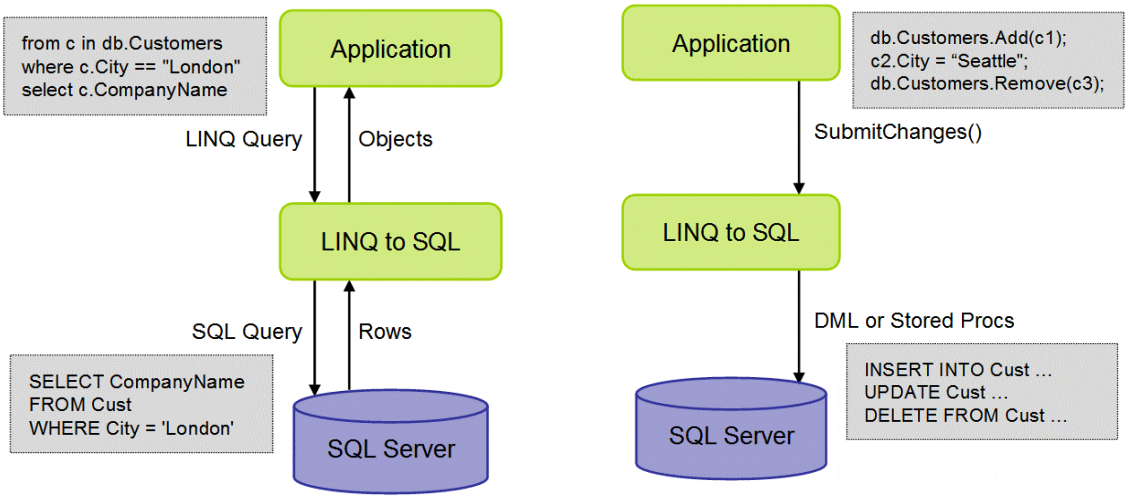
Architektura pro vykonávání dotazů se dá rozdělit do dvou hlavních částí:

- První část znázorněná na obrázku 4(a) provádí dotaz do databáze. LINQ neobsahuje žádný query operátor pro insert, delete nebo update. Když se pošle LINQ dotaz na LINQ to SQL vrstvu, ta ho přeloží na SQL (T-SQL) dotaz. Je to z důvodu zachování nezávislosti na jakémkoliv API a možnosti spolupráce s dalšími verzemi SQL serveru. Výsledek je pak vrácen zpět na LINQ vrstvu a vytvoří objekt implementující rozhraní `IEnumerable <T>`.
- Na obrázku 4(b) je zobrazen způsob provádění DML příkazů nebo procedur. Pokud je použita uložené procedura, tak umožňuje upravit insert, update a delete odpovídajícím způsobem. Toto provádění modifikací se děje čistě prostřednictvím objektového modelu.
  - Insert - vytvoří objet včetně všech potřebných vlastností. Tento objekt přidá do existující kolekce a zavolá metodu `SubmitChange()`.
  - Update - získá objekty, které jsou upraveny a zavolá nad nimi metodu `SubmitChange()`
  - Delete - získá objekty pomocí metody Remove() a odebere je z existující kolekce. Po té zavolá metodu `SubmitChange()`.[17]

3.3.3 Mapování a tvorba modelu

Pro vytvoření modelu máme tři možnosti:

- Objektově relační návrhář - je součástí Visual Studia a dá se použít pro vytvoření modelu z již existující databáze. Vytvoříme novou LINQ to SQL Class s příponou .dbml



(a) Načítání (b) DML

Obrázek 4: Architektura LINQ to SQL [17]

(Database Mapping Language). Okno návrháře je rozdělené do dvou sekcí. V levé větší můžeme přetáhnout již existující tabulky nebo pohledy a zobrazí se i patřičné vazby mezi nimi. Do pravého okna se dají vložit databázové procedury nebo funkce. Použití tohoto nástroje je vhodné pro menší a středně velké databáze. Výsledný model je uložený ve třech samostatných souborech.

- Nástroj pro generování SQLMetal - je nástroj využívající příkazovou řádku k vygenerovaní modelu, vhodné použití pro generovaní modelu u velkých databází. [18]
- Napsání vlastního kódu například pomocí Visual studia. Tento postup může být náchylný k chybám a není doporučený, pokud už existuje databáze.

Kardinalita vztahu

Linq to SQL definuje *AssociationAttribute* atribut, který pomáhá reprezentovat vztahy. Tento atribut se používá většinou ve spojení s `EntitySet<TEntity>` a `EntityRef<TEntity>`, které reprezentují vztah cizího klíče v databázi. Většína vztahů je typu 1:M (one-to-many), ale je možné definovat i vztahy 1:1 (one-to-one) nebo M:N (many-to-many) následujícími způsoby: [19]

- one-to-one - představuje druh vztahu, kde je použitý na obou stranách spojení `EntitySet<TEntity>`.
- many-to-many - v tomto typu vztahu je primární klíč odkazované tabulky (někdy také nazývaná spojovací tabulka) obvykle složený z cizích klíčů z ostatních dvou tabulek. Např. pokud máme tabulku item a warehouse, tak spojovací tabulka by byla location. Linq to

SQL požaduje v tomto případě, aby byly vytvořený tři třídy modelující tento příklad: Item, Warehouse a Location.

Kontext

DataContext je základním objektem pro načtení dat z databáze a jejich opětovného uložení. Použití je podobné jako u objektu Connection z ADO.NET. DataContext je inicializován pomocí zadaného connection stringu. Účelem je přeložení dotazů z objektů do jazyka SQL, provedení v databázi a sestavení objektu z vráceného výsledku. Ve výchozím nastavení je pro načítání dat použitý lazy load. Sleduje změny provedené ve všech načtených entitách a spravuje „identity cache“.

Pro „identity cache“ je použit návrhový vzor Identity map. Pokud je načtený nový řádek z databáze, tak je uložen do tabulky identit podle primárního klíče a je vytvořený nový objekt. Pokud dojde k načtení stejného řádku, který už je uložený v tabulce, tak je vrácený odkaz na původní objekt. Tím je zaručena integrita dat. Aplikace pouze vidí objekt ve stavu, ve kterém byl poprvé načtený. Nové hodnoty pokud jsou jiné, zahodí.

`DataContext` má v sobě zabudovanou podporu pro optimistický souběh tím že automaticky zjišťuje konflikt změn. Dá se nastavit stupeň detekce konfliktu změn při definování třídy entity. Každý sloupec má atribut `UpdateCheck`, který může nabývat tří hodnot:

- Always
- Never
- WhenChanged

Pokud není nastavena, tak výchozí hodnota je Always. To znamená, že kontrola je prováděna vždy. Atribut sloupce má vlastnost `IsVersion` umožňující určit, zda verze dat odpovídá verzi časového razítka odpovídající databázi. Pokud tato verze existuje, tak se použije k porovnání, jestli došlo ke konfliktu.

Pokud tabulka neobsahuje primární klíč, tak je možné data z ní pouze načíst, ale už není možné provést aktualizaci záznamů nad ní.

Obvykle LINQ to SQL aplikace vytváří instanci `DataContext` pro metodu odpovídající logické sadě souvisejících databázových operací. [20]

3.4 Nevýhody současných řešení

Závěrem této části bych zmínil důvod, proč jsou současná řešení nevyhovující a bylo rozhodnuto o jejich nezakomponování do aplikace a proč bylo zvoleno vlastní řešení, které má snadněji řešit a splňovat požadavky na mapování mezi aplikací a databází.

Entity framework je sice komplexní řešení, ale při častých změnách struktury metadat databázových objektů (tabulek, pohledů, uložených procedur) je neustále problematické aktualizovat

model. Pokud už je model jednou vytvořený, tak je problém provést jeho aktualizaci. Musí se celý odstranit a vytvořit znovu. Proto je problematické udržovat model pro různá prostředí (testovací, produkční), protože v nich je zpravidla vždy jiný. Pokud model v aplikaci není totožný se strukturou databáze, tak dojde k jeho pádu a i pádu celé aplikace.

Další nevýhodou je, že se často používají vlastní metody načtení dat (podrobněji popsané v následující kapitole 4), které pracují s dočasnými tabulkami (`create table #item` uložených v tempDB). Někoho může napadnou, proč se místo nich nepoužije tabulková proměnná (`declare @item table`). Ta má však několik nevýhod:

- nemůže obsahovat neklastrovaný index
- nemůže být nad ní vytvořené integritní omezení (constraint)
- nemůže být vytvořena výchozí hodnota (default)
- nejde na ní vytvořit statistiku

Entity framework ve starších implementacích (aktuální verze nebyla vyzkoušená) používá pro získání informací o uložených procedurách nastavení `SET FMTONLY`, které v tomto případě kdy dojde k chybě, o ní neinformuje a požadovanou proceduru do modelu nezahrne.

4 Vlastní řešení

V rámci této práce jsem řešil pouze tvorbu definice datového zdroje. Jak jsou tyto datové zdroje zpracovány střední vrstvou nebo způsob volání z klienta nebylo součástí této práce. Použití na klientské straně bude popsané v kapitole 4.5, stejně tak bude zmíněný i způsob zpracování střední vrstvou aplikace.

Nejdříve popíši obecný princip funkcionality a pak definuji pojmy související s datovým zdrojem a jeho použitím a pokusím se je ilustrovat na příkladech pro snadnější pochopení.

4.1 Obecný popis řešení

Princip fungovaní mapování by se dal přirovnat variantě zmíněné u EF nazvané Database First. Předpokladem je tedy již existující databáze obsahující tabulky, pohledy, procedury a funkce. Protože vytváření probíhá již na základě existující databáze, tak implementace dědičnosti by byla zbytečnou složitostí s minimálním přínosem, proto nebyla pro potřeby implementována. Nevytváří se ani objektový model vhodný např. pro využití *IntelliSense* (je obecný termín pro různé funkce při editaci nebo psaní zdrojového kódu. Umožňuje dokončení kódu, zobrazuje informace o parametrech, členech atd.), protože požadavek na zdroj dat a potřebné sloupce je známý dopředu. Podmínka selekce může mít dva typy:

- pevně daný - jde o podmínku v dotazu `WHERE`, která už může být definovaná přímo v definici metadat datového zdroje
- dynamicky doplněná - tato podmínka je předaná z aplikace jako filtrovací kritérium při sestavení dotazu za běhu aplikace. Např. pokud chceme zobrazit jen parametry konkrétní uložené procedury

Pro přístup do databáze se využívá nativní přístup pomocí `SQLClient`. Výhodou je dobrá implementace v .Net. Pro volání dotazů se používá `SQLCommand` s parametrem `CommandType.StoredProcedure`. Při tomto nastavení dochází k RPC volání. Je to výhodné z pohledu optimalizace, protože pokud již existuje exekuční plán pro proceduru, tak je použitý.

Dalším kladem je i známý výsledek zpracování, protože jako první parametr výsledku volání procedury je návratová hodnota a následují parametry, které mohou být buď vstupní, výstupní nebo vstupní a výstupní. Funkcionalita je obdobná jako u volání metody s návratovou hodnotou v .Net. Princip je nastíněn na krátké ukázce kódu volání metody a zjištění její návratové hodnoty. Do proměnné `@Sev` se nám vrátí číselná hodnota, které slouží k identifikaci výsledku běhu procedury (viz příklad níže).

```
-- create procedure
if object_id('ReturResSp') is not null
  drop procedure ReturResSp
go
```

```
create procedure ReturResSp (@vstup int, @vystup nvarchar(50) output)
as
  declare
    @Sev int = 0
 --implementace
  set @vystup = N'message...'
  set @Sev = 16
return @Sev
go

--method call
declare @Sev int, @Vystup nvarchar (50)

exec @Sev = dbo.ReturResSp 1, @Vystup output
```

Pro načtení vrácených hodnot se používá `DataReader`, kde se na první pozici nastaví návratová hodnota a pak už následují parametry v pořadí tak, jak jsou definovány v proceduře. Pro předání parametru se použije konstrukce např. `item = @item`. Při změně procedury nebo obecně databázových metadat není potřeba řešit sestavení celé aplikace znovu, ale jen se změní definice objektu datového zdroje pro střední vrstvu aplikace.

Pro uživatelsky definovanou metodu načtení dat do datového zdroje je vytvořený indexer, který se používá pro vrácení hodnoty například z řádku a sloupce jedna. Přijde název sloupce, zjistí se číslo sloupce a sáhne se do dat na odpovídající pozici.

Transakční zpracování procedur se řeší podle příznaku u metody definované v metadatech.

4.2 Definice pojmů datového zdroje

V této části jsou definovány pojmy související s datovým zdrojem a na jednotlivých příkladech se je budu snažit popsat. Definice a použití vysvětlím na příkladu „dodacího listu“. Jsou nadefinované dvě tabulky a dvě uložené procedury:

- **dl** tabulka, představuje hlavičku dodacího listu
- **dlradek** tabulka, obsahuje seznam řádků patřících k dodacímu listu
- **expDlSp** uložená procedura pro expedici dodacího listu
- **loadDataDlSp** vlastní metoda načtení dat

```
CREATE TABLE dl(
    cis_dl nvarchar(10) PRIMARY KEY, stav nvarchar(2), cis_zak nvarchar(10),
       zak_dod int, datum_vyt datetime
```

```
)

CREATE TABLE dlradek(
    cis_dl nvarchar(10), typ_radku nvarchar(1), rad_cis int, cis_zakazky
       nvarchar(10), rad_zak int, qty_poz decimal(15,5), qty_exp decimal(15,5),
        exp tinyint
)

CREATE PROCEDURE expDLSp(@cis_dl nvarchar(10), @mes nvarchar(255) output)
as
 declare @Sev int
 set @Sev = 0
 ... -- zpacovani procedury
return @Sev

CREATE PROCEDURE expDLSp(@cis_dl nvarchar(10), @mes nvarchar(255) output)
as
  declare @Sev int
  set @Sev = 0

  create table #user_result (
      jmeno nvarchar(30)
   , prijmeni nvarchar(30)
   , ulice nvarchar(50)
   , psc nvarchar(6)
  )
  ...-- zpacovani procedury
  select prijmeni, jmeno, ulice, psc
  from #user_result

return @Sev
```

Datový zdroj

Datový zdroj reprezentuje objekt, který nese v sobě informaci o tom, jaké tabulky nebo pohledy používá, které vlastnosti (sloupce) z nich obsahuje a které uložené procedury je možné z jeho kontextu zavolat. Ve standardním *databázovém schématu* dbo se nacházejí tabulky, pohledy, procedury a funkce sloužící pro definici datového zdroje. Tabulky pro uložení definice datového zdroje jsou uloženy ve vlastním databázovém schématu core.

Databázové schéma je způsob, jak lze logicky seskupovat dohromady objekty jako jsou např. tabulky, pohledy, procedury atd. Schéma si lze tedy představit jako kontejner pro tyto objekty. K schématu můžeme přiřadit oprávnění přístupu pro přihlášeného uživatele, aby mohl přistupovat jen k objektům ve schématu, ke kterému má přiřazena práva. Při vytvoření nové databáze se nově přidané objekty uloží do výchozího schématu dbo, pokud vytvoření databázové objektu není uvedeno jinak jako např. na příkladu definice tabulky: `CREATE TABLE [core].[item] (..)`

Na první pohled by se mohlo zdát, že datový zdroj by mohl být nahrazený pohledem, proto nejdříve uvedu na krátkých příkladech limity použití pohledu:

- do pohledu není možné předat parametr. Konstrukce podobná definování uložené procedury nebo funkce není možná. Uvedu na příkladu viz níže:

```
   CREATE TABLE item (
    item nvarchar(50) PRIMARY KEY
  , description nvarchar(100)
  , qty int
)

CREATE VIEW itemview @qty int
AS
SELECT item, description, qty from item
WHERE qty > @qty
```

  Řešením je buďto použití *tabulkové funkce*. Je to funkce, která vrací tabulku, nebo se musí zapsat podmínka přímo do WHERE klauzule. Ukázka řešení je na příkladu níže pomocí tabulkové funkce.

```
create function itemFn (@qty INT)
RETURNS TABLE
 AS
 RETURN
 (select item, description, qty
 FROM item WHERE price < @qty)

SELECT * FROM itemFn(100) // call
```

- není možné definovat v pohledu setřídění pomocí klauzule `ORDER BY`. Přesněji je možné použít setřídění dotazu, ale jen ve spojení buď s `TOP` x (kde x definuje počet vrácených setříděných záznamů, tedy TOP 5 vrátí prvních 5 záznamů), nebo pomocí `FOR XML`.

```
ALTER VIEW itemview
AS
```

```
SELECT --TOP 5 - bez pouziti TOP pohled nejde vytvorit, po odkomentovani
    pujde vytvorit
  item, description, qty from item
FROM item
ORDER BY qty
```

- není možné vytvořit pohled nad *dočasnou tabulkou* (temporary table). Dočasná tabulku je uložená v tempdb databází a může být dvojího typu:
  - lokální - lokální tabulka je viditelná pouze tomu, kdo ji vytvořil od okamžiku založení po dobu trvání přihlášení k databázi. Automaticky je smazaná po odhlášení uživatele. Vytváří se pomocí příkazu `CREATE TABLE #itemTmp( item nvarchar(50)...)`
  - globální - je viditelná všem přihlášeným uživatelům od doby jejího vytvoření až do doby, dokud se všichni uživatelé, kteří se na ní odkazují, neodhlásí. Pak je teprve odstraněna. Vytvoří se pomocí příkazu `CREATE TABLE ##itemTmpGl(item nvarchar(9)...)`

```
CREATE TABLE ##itemTmpGl(
    item nvarchar(50) PRIMARY KEY
  , description nvarchar(100)
  , qty int)
)

CREATE VIEW itemTmpGlView
AS
   SELECT item, description, qty
   FROM ##itemTmpGl
```

- nemůže obsahovat pravidla, default a index. Přesněji řečeno index jde vytvořit nad pohledem, ale jen pokud je vytvořeno s `WITH SCHEMABINDING` (v tomto případě ale dochází k zamčení metadat) a to jen za předpokladu, že pohled není složený z více tabulek.
- může obsahovat jen jeden *statement*. Statement je jakýkoliv text, který databázový engine rozpozná jako platný příkaz např. `SELECT`

Výhody použití datového zdroje:

Vlastní metoda načtení dat

Jedná se o uloženou proceduru. Ta se používá pro vrácení svého resultu a jejího spárování na vlastnosti datového zdroje. Toto provázání se provádí zpravidla na vlastnosti typu unbound. Pokud např. vlastní metoda načtení dat vrací 4 sloupce jmeno, prijmeni, ulice a psc a máme v

Tabulka 1: Spárování vlastnosti s navráceným resultem z vlastní metody načtení dat

Vlastnost	Index
Jmeno	2
Prijmeni	1
Ulice	3
Psc	4

datovém zdroji definované 4 unboud vlastnosti Jmeno, Prijmeni, Ulice a Psc, vytvoří se pro spárování entice, kde se řekne, že první sloupec z výstupu procedury se spáruje na jakou vlastnost podle indexu (viz. tabulka 1). Výhodou této možnosti je, že ne vždy je možné potřebná data dostat jen z tabulky nebo pohledu (nebo použitím spojení či přes cross/outer apply). Někdy je potřeba na základě vstupních parametrů dotáhnout data z jiných struktur nebo na základě podmínky provést jiné výpočty. Často je potřeba také v rámci zpracování nebo dotažení potřebných dat provést dynamický dotaz do jiné databáze, což v rámci pohledu není možné. Také je možné použít i dočasné tabulky pro uložení mezi výsledku při zpracování.

Tato možnost je ale až od verze SQL serveru 2012. Už předchozí verze SQL serveru přišla s možností vrácení seznamu sloupců z uložené procedury pomocí nastavení `SET FMTONLY`. Tento způsob ale není moc použitelný, protože se často využívá ve vlastních metodách načtení dat pro vrácení výsledku dočasných tabulek, kde dotaz nedoběhne korektně. Od verze SQL serveru 2012 a novější se používá pro vrácení seznamu dynamická funkce `sys.dm_exec_describe_first_result_set()`.

Typ vlastnosti – podkolekce

Vytvořený zdroj dat se dá použít jako vlastnost v jiném zdroji dat. Určí se podmínka spojení, podle které bude podřízená kolekce filtrována.

Typ vlastnosti - derived

Jde o vlastnost, kde může být použitý poddotaz, skalární funkce, jiná vlastnost nebo konstanta. V toto případu vlastnost není vázaná ke konkrétnímu sloupci z tabulky nebo pohledu definovaného ve zdroji dat. Vysvětlím možnosti na následujícím příkladu: Mám vlastnost pojmenovanou `DerExistuje`:

- poddotaz - *(case when exists (select 1 from dlradek l where l.cis_dl = CisDl and l.typ_radku = N'Z') then 1 else 0 end)* v příkladu je použita i vlastnost `CisDl`, která se pak při vyhodnocení nahradí skutečnou hodnotou odpovídající této vlastnosti.
- skalární funkce - dbo.NazevFunkce(l.cis_dl)
- jiná vlastnost - pokud například máme definované dvě vlastnosti `MnozSkladem` a `MnozstviExp`, které jsou obě celá čísla, tak pokud použiji zápis `MnozSkladem - MnozstviExp` a hodnoty by byly 200 a 50, tak výsledek by se zobrazil po vyhodnocení jako 150.

Typ vlastnosti unbound

Jde o vlastnost, která není přiřazena žádnému sloupci z tabulky nebo pohledu. Je u ní definovaný pouze název a jakého je datového typu. Její obsah bude vložen až za běhu aplikace například pomocí vlastní metody načtení dat.

Obecně v různých datových zdrojích mohou být použité stejné tabulky nebo pohledy. Pokud v rámci jednoho datového zdroje potřebujeme přidat novou vlastnost, tak v případě pohledu by byla potřeba vytvořit redundantní kopii s přidáním jednoho sloupce, anebo by byla potřeba ve všech případných použití daného view provést úpravu v aplikaci, kde se používá.

Přesněji to vysvětlím na příkladu: Máme již dříve zmíněné tabulky dl a dlradek, nad kterými máme vytvořený datový zdroj *DodociList*, který používají všichni zákazníci. Jeden ze zákazníků přijde s požadavkem, že bude potřebovat evidovat jaká sériová čísla byla expedovaná s jakou zakázkou. To se pomocí datového zdroje udělá snadno. Vytvoříme nový datový zdroj pro zákazníka, kde připojíme tabulku se sériovými čísly a přidáme požadované vlastnosti do datového zdroje. Pak už jen řekneme v definici, že zákaznický datový zdroj *ZakazniDodaciList* rozšiřuje původní datový zdroj *DodaciList*.

Konzistence datového zdroje

Konzistence definice datového zdroje se kontroluje na dvou úrovních. První je při samotném mazání různých elementů datového zdroje a druhá při samotném uložení datového zdroje. Podrobně teď popíši na příkladech, jak kontrola probíhá pro jednotlivé části:

Mazání jednotlivých součástí datového zdroje:

- datový zdroj - ten jde smazat jen za podmínky, že není použitý jako vlastnost typu podkolekce v jiné definici datového zdroje
- vlastnost - vlastnost definována v rámci datového zdroje jde smazat jen tehdy, pokud není použita v rámci jiné vlastnosti typu derived, nebo vlastnost se nenachází v definici spárování u vlastní metody načtení dat
- třída vlastnosti - jde smazat, pokud není použita ve vlastnosti datového zdroje. Tato kontrola se provádí přes všechny existující datové zdroje
- tabulka (pohled) - není možné smazat, pokud existuje alespoň jedna vlastnost datového zdroje, která se na ni odkazuje

Zapsaní datového zdroje (ukončení editace datového zdroje):

Při uložení změn datového zdroje se provedou následující kontroly. Pokud nejsou splněny, tak datový zdroj není možné uložit a tím pádem ani vyexportovat a přenést:

- existence tabulky/pohledu - zkontroluje se, jestli v databázi existuje tabulka nebo pohled použitý v definice datového zdroje
- existence procedury - provede se kontrola, jestli procedura použitá v rámci datového zdroje existuje v databázi, jestli sedí počet jejich parametrů, datový typ a informace o typu parametru (vstupní, výstupní)
- existence vlastnosti - zkontroluje se, jestli vlastnost odkazující se na konkrétní sloupec v tabulce se v definici tabulky v databázi opravdu nachází, tj. jestli ho např. někdo mezitím neodstranil

Těmito kontrolami bude zajištěna konzistence datového zdroje s použitými metadaty v databázi.

Ostatní vlastnosti budou zmíněné v následující kapitole 4.3

4.3 Funkční analýza

Primárně je nástroj určený pro administrátory, analytik se může v případě potřeby podívat do definice datového zdroje, ale nebudu mu umožněna jeho úprava.

1. **Třída vlastnosti**: protože se používají (vytvářejí) vlastní databázové datové typy, tak se pro ně vytvoří zároveň i třída vlastnosti. V ní se definuje např. velikost, popisek, maska a další vlastnosti. Při použití konkrétního datového typu v jiné tabulce, pohledu je možné znovu použít třídu vlastností k dotažení výchozích hodnot a případně změnit některou vlastnost, pokud bude potřeba.

   Tabulka: MdPropertyClass

   (a) Vložení nové třídy vlastnosti
   (b) Aktualizace třídy vlastnosti
   (c) Smazání třídy vlastnosti – lze smazat, pokud už není nikde použita
   (d) Načtení seznamu tříd vlastnosti

2. **Import definice datového zdroje z XML souboru** – po vytvoření definice v testovacím prostředí se přenese a naimportuje do produkčního prostředí.

   Tabulky: *

   (a) Import definice datového zdroje z XML souboru – pokud pro název definice datového zdroje existuje již záznam, tak dojde v prvním kroku ke smazání všech částí a opětovnému vložení.

3. **Export definice datového zdroje do XML souboru** - používá se pro přenos z testovacího prostředí do produkčního.

   Tabulky: *

(a) Export vybrané definice datového zdroje podle názvu do XML souboru

4. **Definice datového zdroje**

   Tabulka: MdCollection

   (a) Vytvoření definice datového zdroje (jedná se o deklaraci datového zdroje)

   (b) Smazání definice datového zdroje – smazání je možné jen pokud je záznam v editačním režimu (dojde k vymazání všech souvisejících definic z ostatních tabulek)

5. **Editační režim** - slouží k úpravě již existující definice datového zdroje a k uchování informace o provádění úpravy nad datovým zdroje uživatelem. Někdo by si mohl položit otázku, proč není na začátku vytvořena databázová transakce a po ukončení úprav nebo vytvoření nového zdroje dat potvrzena, případně provedené změny vráceny. Je to ze dvou důvodů:

   - při vytváření datového zdroje je požadavek na průběžné ukládání dílčích částí např. přidání metody
   - potřebujeme ošetřit situaci, aby pokud by došlo k chybě média nebo jiné chybě vedoucí následovné k obnově databáze, aby nebyla rozdělaná práce ztracena. V tomto případě by nám nepomohlo ani použití *savepoint*u, protože i v tomto případě je neukončená transakce vrácena. Savepoint je příkaz SQL serveru, umožňující se vrátit k určitému bodu zpracování transakce při provedení *rollback*. Rollback je kompletní zrušení změn provedených od zahájení transakce.

   Tabulka: MdLock

   (a) Nastavení editačního režimu – pokud je vytvořená nová definice datového zdroje, tak se automaticky vloží i nový záznam do příslušné tabulky pod přihlášeným uživatelem s názvem definice dat. Případně je možné záznam přepnout ručně do editačního režimu za předpokladu, že daný záznam již není upravovaný jiným uživatele. Záznam, který je editován, není možné vyexportovat. Není možné, aby daný záznam editovalo zároveň více uživatelů. Tím je zabráněno vzniku nekonzistentního datového zdroje.

   (b) Vypnutí editačního režimu – provede se kontrola konzistence datového zdroje a pokud je v pořádku, tak dojde k odstranění záznamu z tabulky pro příslušného uživatele

6. Definice zdroje dat datového zdroje (tabulky, pohledy) - pokud se dále bude hovořit v souvislosti s datovým zdrojem dat o tabulce, tak to může být databázová tabulka nebo pohled. V tomto směru se jedná o ekvivalentní objekty.

   Tabulka: MdTable

(a) Vytvoření zdroje – zvolení primárního zdroje dat. Ten může být jen jeden (tabulka nebo pohled). Dále je zde možné přidat sekundární zdroje (tabulky, pohledy), u kterých se definuje druh spojení (inner nebo left join).

(b) Úprava zdroje

(c) Smazání zdroje

(d) Vrácení seznamu pohledů nebo tabulek pro přidání zdroje

7. **Definice vlastnosti zdroje** - vlastnosti (property)

   Rozlišují se tři základní druhy vlastností:

   - *bound* - jde o konkrétní sloupec z tabulky nebo pohledu definovaného ve zdroji dat
   - *derived*
   - *unbound*

   Speciálním typem vlastnosti je *podkolekce*. Tato vlastnost nám umožňuje odkazovat se na jinou definici datového zdroje.

   Taublka: MdProperty

   (a) Vytvoření vlastnosti - přidání jednotlivých sloupců z definice dat datového zdroje. Při použití třídy vlastnosti se doplní předdefinované hodnoty pro danou vlastnost, ale ty je možné změnit.

   (b) Změna vlastnosti - změna některých vlastností.

   (c) Smazání vlastnosti - odstranění vlastnosti je možné, pokud je definice v editačním režimu.

8. **Definice metod datového zdroje** - (uložená procedura)

   - Po výběru existující procedury se načte z metadat seznam jejich parametrů, datové typy a příznak, jestli je vstupní, výstupní nebo výstupní a výstupní a uloží se seznam parametrů. Ten je možné v případě změny procedury (změna počtu parametru, typ parametru nebo datového typu) aktualizovat, ale jen v případě, že je objekt definice zdroje v editačním režimu.
   - Další vlastností, co se dá u definice metod procedury nastavit, je jestli poběží v transakci nebo ne. Tímto je řešený případný souběh.

   Všechny úpravy s metodami je možné provádět pouze tehdy, pokud je definice datového zdroje nastavena na editační režim. Informace o metadatech uložených procedur se berou ze standardního databázového schématu dbo.

   Tabulky: MdMetod, MdMethodParameter

(a) Přidání procedury - možnost přidání jen existující procedury

(b) Aktualizace procedury - provede se synchronizace s metadaty uložené procedury v databázi (počet parametrů, datové typy a typ parametru)

(c) Odebrání procedury

(d) Vrácení seznamu uložených procedur - pro výběr se načte seznam všech procedur nacházejících se v databázi

9. **Definice vlastní metody načtení dat** *Tabulky*: MdMetod, MdMethodParameter, MdMethodOutputSet

   (a) Přidání vlastní metody načtení dat - včetně vytvoření mapování z reusultsetu procedury na vlastnosti datového zdroje

   (b) Aktualizace vlastní metody načtení dat - provedení synchronizace parametrů

   (c) Odebrání vlastní metody načtení dat – jen v editačním režimu

   (d) Vrácení seznamu procedur – pro výběr se načte seznam všech procedur nacházejících se databází pod standardním schématem dbo

4.4 Konceptuální model

Schéma konceptuálního modelu je zobrazeno na obrázku 5

4.4.1 Popis tabulek

- **MdCollection** - zde jsou uloženy deklarace o datovém zdroji.
- **MdTable** - obsahuje seznam všech tabulek nebo pohledů a jejich spojení mezi sebou.
- **MdProperty** - obsahuje seznam všech sloupců (vlastností), ke kterým je možné v rámci datového zdroje přistupovat.
- **MdPropertyClass** - obsahuje definici vlastností přidružených k odpovídajícímu vytvořenému vlastnímu databázovému typu.
- **MdMethod** - obsahuje název procedury a informace o tom, zda daná uložená procedura bude spuštěna v transakci.
- **MdMethodParameter** - seznam parametrů procedury.
- **MdMethodOutputSet** - v případě definice vlastní metody načtení dat se zde nachází informace o přiřazení resultu na odpovídající vlastnosti
- **MdLock** - slouží k uchování informace o uživateli, který upravuje odpovídající datový zdroj

4.5 Použití datového zdroje v aplikaci

Na jednoduchém příkladu definovaném v kapitole 4.2 si ukážeme, jak se s výsledným datovým zdrojem v aplikaci pracuje. Datový zdroj se jmenuje **DodaciList**.

Pokud potřebuji zavolat proceduru deklarovanou v datovém zdroji, tak se to udělá následujícím způsobem:

```
Public Async Function Expedice() As Task(Of EventHandlerResponse)
  Dim resp As EventHandlerResponse = Nothing
  Dim respState As InduStream.Shared.Responses.IdoMethodCallResponse = Nothing

  respState = Await Me.CallIDOMethod("DodaciList", "expDlSp", New Object() {"V(
      vCisDL)", "RV(vMessage)"})

  If respState.Severity > 0 Then
    Me.MainForm.ShowMsgBox(FrmITeuroTSDMainForm.FormatMsgBoxString(vMessage.
        Value), Nothing)
    Return New EventHandlerResponse(False)
  End If

  resp = New EventHandlerResponse(True)
  Return resp
End Function
```

Metoda `CallIdoMethod` jako první parametr má název datového zdroje, druhý název procedury a poslední je seznam parametrů volané procedury. U seznamu parametrů klíčové slovo `V(..)` znamená, že se jedná o vstupní parametr a `RV(...)` výstupní parametr. Výsledek volání procedury se uloží do proměnné `respState`. V případě vrácení chybového stavu, se zobrazí informativní oznámení v aplikaci s popisem chyby vrácené ve výstupním parametru `vMessage`.

Pokud je datový zdroj nastavený ve vlastnostech v gridu, tak se jeho inicializace a vrácení dat provede při otevření formuláře. Stejná funkcionalita by se vyvolala i v případě, pokud by se aktualizace provedla např. přes tlačítko:

```
Async Function StdInit() As Task(Of EventHandlerResponse)
  Dim resp As InduStream.Shared.Responses.IdoPrimaryCacheLoadResponse

  resp = Await Me.IdoPrimaryCacheLoad("DodaciList", "CisDL, Stav, CisZak,
      DatumVyt", Nothing, -1)

  If resp.Severity <> 0 Then
```

```
    Me.MainForm.ShowMsgBox(FrmITeuroTSDMainForm.FormatMsgBoxString("Data load
        error"), Nothing)
    Return New EventHandlerResponse(False)
  Else
    DtDodList = resp.PrimaryCache.Tables(0)
  End If

  Return New EventHandlerResponse(True)
End Function
```

Metoda `IdoPrimaryCacheLoad` má jako první parametr název datového zdroje, následuje pak seznam vlastností, třetí parametr je filtr a poslední parametr udává maximální počet záznamů, které budou vráceny. V tomto příkladu „-1“ znamená výchozí hodnotu. Počet vrácených záznamů se tedy bere podle nastavení střední vrstvy. Pokud by zde bylo uvedené jiné číslo větší než „0“, tak by se vrátil počet záznamů odpovídajících tomuto číslu.

Použití vlastní metody načtení dat:

```
Public Async Function ReloadDl() As Task(Of EventHandlerResponse)
    Await Me.DataSourceLoadCLM("loadDataDlSp", New Object() {ThisForm.Variables
        ("vCisDL").Value})

    Return New EventHandlerResponse(True)
  End Function
End Function
```

V datovém zdroji `DodaciList` je definovaná vlastní metoda načtení dat `loadDataDlSp`. Metoda `DataSourceLoadCLM` má jako první parametr název vlastní metody načtení dat a jako druhý parametr filtr hodnot, podle kterých je možné záznamy odfiltrovat.

Skládání dotazu na střední vrstvě

V případě, že se pro načtení dat použije datový zdroj, tak se výsledný dotaz složí do textového řetězce. Pro lepší popis složení výsledného dotazu je znázorněný fragment definice datového zdroje v tabulce 2.

Chceme spojit dvě tabulky `dl` a `dlradek` a z nich chceme zobrazit informace o čísle dodacího listu (CisDL). Z řádku pak informace o požadovaném množství (QtyPoz), expedovaném množství (QtyExp), kolik ještě zbývá množství vyexpedovat (derived vlastnost DerQtyRem) a zjištění informace o dodacím listu (DerInfo).

Vezme se seznam vlastností (sloupců) a přidá se hned za klíčové slovo `SELECT`. Pokud je v rámci definice vlastnosti použita derived vlastnost a ta v sobě nese informaci o jiné vlastnosti

Tabulka 2: Poskládání dotazu z definice datového zdroje

Tabulka	TabAlias	DbCol	Spojeni	TypSpoj	Vlastnost	VlastHod
dl	d	-	-	3	-	-
dlradek	dl	-	d.cis_dl = dl.cis_dl	1	-	-
dl	d	cis_dl	-	-	CisDl	-
dl	d	-	-	-	DerInfo	dbo.TestFn(CisDl)
dlradek	dl	qty_poz	-	-	QtyPoz	-
dlradek	dl	qty_exp	-	-	QtyExp	-
dlradek	dl	-	-	-	DerQtyRem	QtyPoz - QtyExp

např. `QtyPoz - QtyExp`, tak se daná vlastnost nahradí odpovídajícím sloupcem z databáze, který je definovaný u vlastnosti. V tomto případě se `QtyPoz` nahradí v dotazu na `dl.qty_poz`.

Typ spojení 3 udává informaci o tom, že daná tabulku je primární zdrojem dat a bude hned do výsledného dotazu doplněná za klauzuli `FROM`.

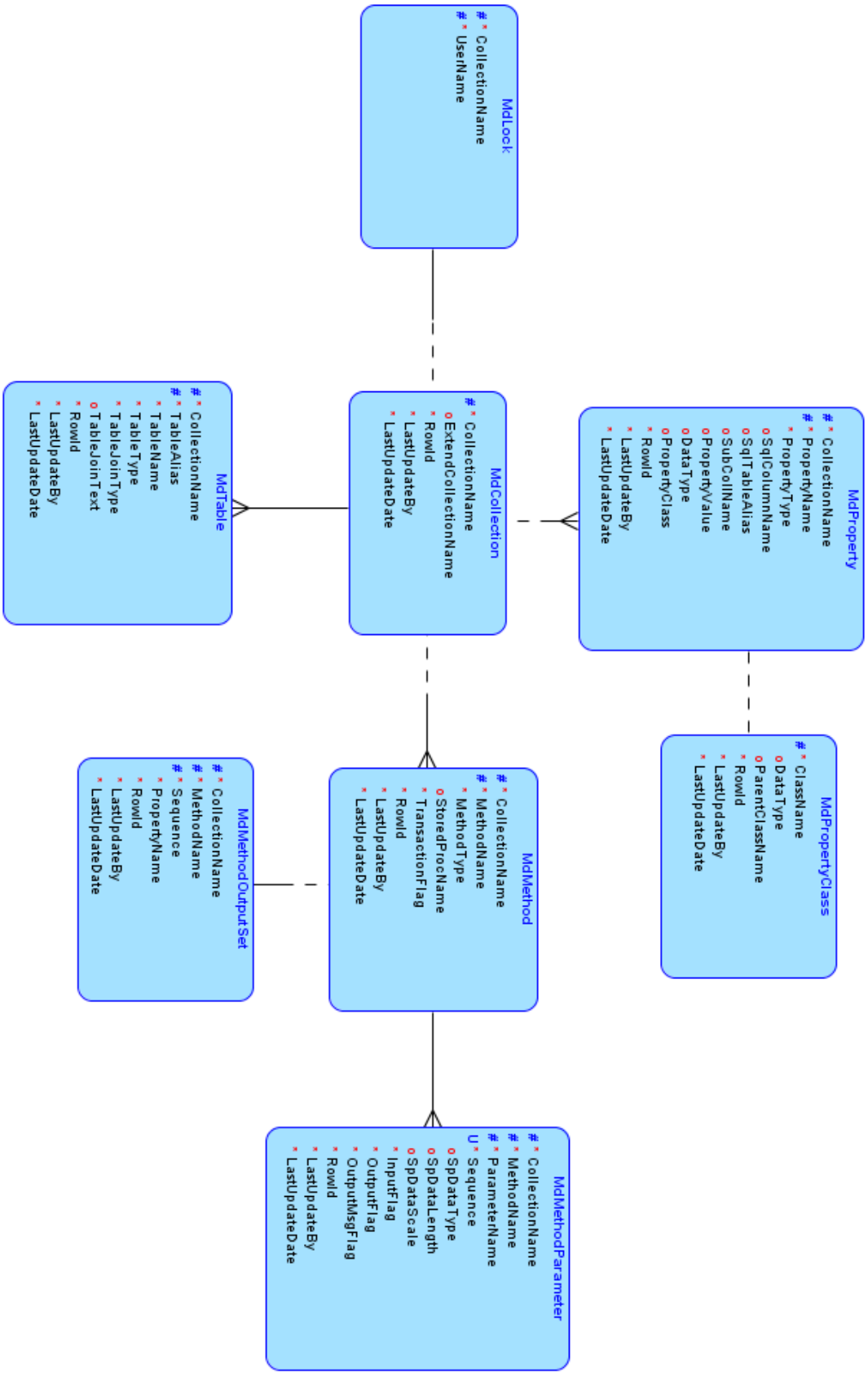
Spojení nám říká, jak jsou tabulky mezi sebou propojeny. V tomto příkladu typ spojení „1“ udává informaci o tom, že tabulky budou spojené pomocí `LEFT OUTER JOIN` a doplní se jen podmínka spojení ze sloupce Spojeni. Výsledný sestavený dotaz, který se pošle do databáze, je znázorněný na příkladu níže.

```
SELECT
    d.[cis_dl] [CisDL]
  , dbo.TestFn(d.cis_dl) [DerInfo]
  , dl.[qty_poz] [QtyPoz]
  , dl.[qty_ext] [QtyExp]
  , (dl.qty_poz - dl.qty_exp) [DerQtyRem]
FROM [dl] d
  LEFT OUTER JOIN [dlradek] dl ON d.cis_dl = dl.cis_dl
```

Zavolání dotazu se provede pomocí `SqlCommand`. Jako první parametr se přidá severita, protože ta je potřeba vždycky vrátit (používá se k vyhodnocení výsledku zpracování dotazu). Nad definovaným `SqlCommand` se zavolá `ExecuteReader()`. Na readeru se ověří, jestli má řádky. Pokud ano, tak se provede čtení. Zavolá se metoda `GetValues(Object())`která vrátí řádek, a ten se uloží do vlastního objektu. Čtení probíhá do doby, dokud je co číst, pak se zvedne interní proměnná sloužící k signalizaci, že existuje další result v readeru. Pokud existuje, tak probíhá zase čtení. Tato varianta se dá použít pro zpracování příkazu `SELECT`, ale s tím rozdílem, že není potřeba řešit přemapování.

Obdobné je to s použitím vlastní metody načtení dat, ale v tomto případě je potřeba procházet přes jednotlivé buňky, protože `GetValue()` je špatně zaobalená metoda, která spouští pořád statistiky. Provádí se iterace přes všechny sloupce v řádku a kontroluje se vůči konverznímu poli. V tomto případu vím, že sloupec, který se mi vrátil z databáze na první pozici, se má nastavit na pozici pět. Pokud se vrátí hodnota „-1“, tak to znamená, že sloupec není použitý. Do řádku se

pak na správnou pozici vloží odpovídající hodnota. Takto vzniklý result row už je přemapovaný vůči requestu.



Obrázek 5: Konceptuální model

5 Závěr

V této bakalářské práci bylo mým prvotním úkolem porovnat existující řešení pro objektové relační mapování pro platformu .Net.

Práci jsem začal studiem problematiky objektově relačního mapování a jakým způsobem přistupují tři zvolené rámce k jejímu řešení. Postupně jsem prošel možnosti mapování a vyzkoušel jsem si je na jednoduchých příkladech pro snadnější pochopení celého konceptu fungování daného rámce. Tento postup jsem zvolil proto, abych pak mohl v další části lépe popsat, proč existující řešení nejsou vhodná a bylo rozhodnuto o vytvoření vlastního řešení pro mapování mezi databází a aplikací.

Hlavním cílem této práce bylo vytvoření aplikace pro definici datového zdroje, který se v aplikaci využívá nejen k získávání požadovaných dat z databáze a zobrazení na klientovi, ale i pro zpětné uložení změn do databáze. S takto vzniklým programem je možné nejen vytvořit datový zdroj, ale umožňuje již vytvořenou definici vyexportovat do XML souboru a přenést ji v této formě do jiného prostředí, kde je možné ji naimportovat a začít používat.

Tímto byla značně usnadněna a zjednodušena tvorba definic a dána možnost vizuální kontroly pro snadnější dohledání a odhalení případné chyby v ní. Vizualizace definice datového zdroje usnadňuje a zkracuje možnost dohledání, odkud se v rámci datového zdroje data načítají a poslouží k rychlejšímu zpracování případného požadavku na změnu či rozšíření ze strany zákazníka.

Na základě struktury definice datového zdroje se pomocí vlastní metody načtení dat načtou potřebné informace z datového zdroje o názvu sloupce, datovém typu a na základě nich se sestaví za běhu buď T4 třída nebo bussines objekt, se kterým se pak bude dále pracovat v aplikaci.

Jedním z následujících plánovaných rozšíření programu bude přidání dalších vlastností při definování třídy vlastností (popisek, možnost omezení počtů znaků u textových řetězců, ...) a s tím související rozšíření vlastností. Protože při tvorbě aplikace byla pozornost zaměřena převážně na implementaci všech požadavků, tak je možné, že bude následovat zaměření se na ergonomičtější práci s GUI programu.

Při tvorbě této práce jsem se podrobněji seznámil s různými technikami přístupu z .Net k databázi a možnostmi různých způsobů, jednak jak načítat informace z databáze, tak i jejich zpětný zápis do perzistentního uložiště. Dále jsem si rozšířil již získané zkušenosti s prací s XML o nové poznatky jak při vytváření samotné definice XML tak i jeho zpracování.

Během psaní aplikace jsem narazil na různé problémy. Jako největší problém, který jsem chtěl vyřešit, bylo zjištění resultu u vlastní metody načtení dat a usnadnit tak vytvoření pole pro spárování mezi vlastnostmi a vrácenými sloupci z uložené procedury. Ale narazil jsem na problémy, protože `SET FMTONLY` neumí pracovat s dočasnými tabulkami. Stejná situace je i v doporučeném náhradním řešení v podobě `sys.dm_exec_describe_first_result_set()`, kde jsem narazil i na další nepříjemnost v případě volání dynamic SQL, kde je potřeba použít

WITH RESULT SETS UNDEFINED. Při jejich řešení jsem nasbíral zajímavé zkušenosti a dovednosti, které uplatním v další své pracovní činnosti.

Literatura

[1] FOWLER, Martin. *Patterns of Enterprise Application Architecture.* [s.l.] : Addison Wesley, 2002. ISBN 0321127420

[2] FUSSELL, Mark. *Foundations of Object-Relational Mapping* [online]. 1997-07-03, [cit. 2018-02-04]. Dostupné z: http://markfussell.emenar.com/blog/object-relational/

[3] BAUER, Christian. *Hibernate An object relational-mapping (ORM) library for Java* [online]. 2005-10-16, [cit. 2017-26-11]. Dostupné z: https://sourceforge.net/p/hibernate/mailman/hibernate-devel/thread/46AE220C-2654-4FA5-A5FE-C0BB5F25092F@jboss.com/

[4] *NHibernate* [online]. 2018-03-16, [cit. 2018-03-18]. Dostupné z: https://en.wikipedia.org/wiki/NHibernate

[5] *NHibernate Reference Documentation* [online] [cit. 20172011]. Dostupné z: http://nhibernate.info/doc/nh/en/index.html

[6] *Entity Framework* [online]. 2017-18-11, [cit. 2017-23-11]. Dostupné z: https://en.wikipedia.org/wiki/Entity_Framework

[7] *Entity Framework Overview* [online]. 2017-03-30, [cit. 2017-22-11]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/overview

[8] JAVIN, Paul. *Top 20 Hibernate Interview Questions for Java J2EE Programmers* [online]. [cit. 2017-24-11]. Dostupné z: http://www.java67.com/2016/02/top-20-hibernate-interview-questions.html

[9] *How Entity SQL Differs from Transact-SQL* [online]. 2017-03-30, [cit. 2017-24-12]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/language-reference/how-entity-sql-differs-from-transact-sql

[10] *QueryView Element (MSL)* [online]. [cit. 2017-12-26]. Dostupné z: https://msdn.microsoft.com/en-us/library/cc716798(v=vs.100).aspx

[11] *EntityClient Provider for the Entity Framework* [online]. 2017-03-30, [cit. 2017-12-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/entityclient-provider-for-the-entity-framework

[12] *Entity SQL Language* [online]. 2017-03-30, [cit. 2017-12-26]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/language-reference/entity-sql-language

[13] *LINQ to Entities* [online]. 2017-03-30, [cit. 2017-12-27]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/ef/language-reference/linq-to-entities

[14] *Auto-Compiled LINQ Queries (Entity Framework June 2011 CTP)* [online]. 2011-06-30, [cit. 2017-12-27]. Dostupné z: https://blogs.msdn.microsoft.com/efdesign/2011/06/30/auto-compiled-linq-queries-entity-framework-june-2011-ctp/

[15] *Comega* [online]. 2004-04-08, [cit. 2017-12-27]. Dostupné z: https://www.microsoft.com/en-us/research/project/comega/?from=http%3A%2F%2Fresearch.microsoft.com%2Fcomega%2F

[16] WARREN, Matt. *The Origin of LINQ to SQL* [online]. 2017-05-31, [cit. 2017-23-12]. Dostupné z: https://blogs.msdn.microsoft.com/mattwar/2007/05/31/the-origin-of-linq-to-sql/

[17] GHOSH, Wriju. *LINQ to SQL : Execution Architecture* [online]. 2007-12-18, [cit. 2018-01-02]. Dostupné z: https://blogs.msdn.microsoft.com/wriju/2007/12/18/linq-to-sql-execution-architecture/

[18] *SqlMetal.exe (Code Generation Tool)* [online]. 2017-03-30, [cit. 2018-01-08]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/tools/sqlmetal-exe-code-generation-tool

[19] *How to: Map Database Relationships* [online]. 2017-03-30, [cit. 2018-02-06]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/sql/linq/how-to-map-database-relationships

[20] KULKARNI, Dinesh, BOLOGNESE, Luca, WARRENT, Matt, HEJLSBERG, Anders, Kit George. *LINQ to SQL: .NET Language-Integrated Query for Relational Data* [online]. March 2007, [cit. 2018-23-12]. Dostupný z: https://msdn.microsoft.com/en-us/library/bb425822.aspx